CSC 372, Spring 2015
Assignment 8
Due: Thursday, April 23 at 23:59:59

Use SWI Prolog!

Use SWI Prolog for this assignment. On lectura that's `swipl`.

Use the tester!

Don't just "eyeball" your output—use the tester! I'll be delighted to help you with using the tester and understanding its output. However, I won't have any sympathy for those who fail tests during grading simply because they didn't want to bother with the tester

Make symbolic links for `a8` and `t` in your assignment 8 directory on lectura, for easy access to the tester and the data files. (And, the tester assumes the `a8` symlink is present.) See the assignment 5 write-up for how-to details.

About the `if-then-else` structure (`->`) and disjunction (`;`)

The rules for `if-then-else` and disjunction are the same as for assignment 7.

Problem 1. (2 points) `splits.pl`

This problem reprises `splits.hs` from assignment 2. In Prolog it is to be a predicate `splits(+List,-Split)` that unifies `Split` with each "split" of `List` in turn. Example:

```
?- splits([1,2,3],S).
S = [1]/[2, 3] ;
S = [1, 2]/[3] ;
false.
```

Note that `Split` is not an atom. It is a structure with the functor `/`. Observe:

```
?- splits([1,2,3], A/B).
A = [1],
B = [2, 3] ;
A = [1, 2],
B = [3] ;
false.
```

Here are additional examples. Note that splitting a list with less than two elements fails.

```
?- splits([],S).
false.

?- splits([1],S).
false.

?- splits([1,2],S).
S = [1]/[2] ;
false.
```

```
?- atom_chars('splits',Chars), splits(Chars,S).
Chars = [s, p, l, i, t, s],
S = [s]/[p, l, i, t, s] ;
Chars = [s, p, l, i, t, s],
S = [s, p]/[l, i, t, s] ;
Chars = [s, p, l, i, t, s],
S = [s, p, l]/[i, t, s] ;
Chars = [s, p, l, i, t, s],
S = [s, p, l, i]/[t, s] ;
Chars = [s, p, l, i, t, s],
S = [s, p, l, i, t]/[s] ;
false.
```

My solution uses only two predicates: `append` and `\==`.

Problem 2. (3 points) `repl.pl`

Write a predicate `repl(?E, +N, ?R)` that unifies `R` with a list that is `N` replications of `E`. If `N` is less than 0, `repl` fails.

```
?- repl(x,5,L).
L = [x, x, x, x, x].

?- repl(1,3,[1,1,1]).
true.

?- repl(X,2,L), X=7.
X = 7,
L = [7, 7].

?- repl(a,0,X).
X = [].

?- repl(a,-1,X).
false.
```

Problem 3. (5 points) `pick.pl`

Write a predicate `pick(+From, +Positions, -Picked)` that unifies `Picked` with an atom consisting of the characters in `From` at the zero-based, non-negative positions in `Positions`.

```
?- pick('testing', [0,6], S).
S = tg.

?- pick('testing', [1,1,1], S).
S = eee.

?- pick('testing', [10,2,4], S).
S = si.

?- between(0,6,P), P2 is P+1, pick('testing', [P,P2], S),
writeln(S), fail.
te
es
st
ti
```

```
in
ng
g
false.

?- pick('testing', [], S).
S = ''.
```

If a position is out of bounds, it is silently ignored. My solution uses `atom_chars`, `findall`, `member`, and `nth0`.

Problem 4. (9 points) `polyperim.pl`

Write a predicate `polyperim(+Vertices,-Perim)` that unifies `Perim` with the perimeter of the polygon described by the sequence of Cartesian points in `Vertices`, a list of `pt` structures.

```
?- polyperim([pt(0,0),pt(3,4),pt(0,4),pt(0,0)],Perim).
Perim = 12.0.

?- polyperim([pt(0,0),pt(0,1),pt(1,1),pt(1,0),pt(0,0)],Perim).
Perim = 4.0.

?- polyperim([pt(0,0),pt(1,1),pt(0,1),pt(1,0),pt(0,0)],Perim).
Perim = 4.82842712474619.
```

The polygon is assumed to be closed explicitly, i.e., **assume** that the first point specified is the same as the last point specified.

There is no upper bound on the number of points but at least four points are required, so that the minimal path describes a triangle. (Think of it as ABCA, with the final A "closing" the path.) If less than four points are specified, a message is produced:

```
?- polyperim([pt(0,0),pt(3,4),pt(0,4)],Perim).
At least a four-point path is required.
false.
```

This is not a course on geometric algorithms so keep things simple! Calculate the perimeter by simply summing the lengths of all the sides; don't worry about intersecting sides, coincident vertices, etc.

Be sure that `polyperim` produces only one result.

Implementation notes

Consider writing a predicate `pair(+List,-Pair)`:

```
?- pair([a,b,c,d],Pair).
Pair = [a, b] ;
Pair = [b, c] ;
Pair = [c, d] ;
false.
```

Use `append` to write `pair`. Then use `pair` with `findall`, `sumlist`, and a predicate that computes the length of a line between two points.

Note that `is/2` supports `sqrt`:

```
?- Y = 3, X is sqrt(Y**2+3).
Y = 3,
X = 3.4641016151377544.
```

Problem 5. (18 points) `switched.pl`

This problem is a reprise of `switched.rb` from assignment 5. `a8/births.pl` has a subset of the baby name data, represented as facts. Here are the first five lines:

```
% head -5 a8/births.pl
births(1950,'Linda',f,80437).
births(1950,'Mary',f,65461).
births(1950,'Patricia',f,47942).
births(1950,'Barbara',f,41560).
births(1950,'Susan',f,38024).
```

`births.pl` only holds data for 1950-1959. Names with less than 70 births are not included.

Your task is to write a predicate `switched(+First,+Last)` that prints a table much like that produced by the Ruby version. To save a little typing, `switched` assumes that the years specified are in the 20th century.

```
?- switched(51,58).
            1951  1952  1953  1954  1955  1956  1957  1958
Dana        1.19  1.20  1.26  1.29  1.00  0.79  0.67  0.64
Jackie      1.40  1.29  1.14  1.13  1.11  0.94  0.72  0.57
Kelly       4.23  2.74  3.73  2.10  2.32  1.77  0.98  0.51
Kim         2.58  1.82  1.47  1.08  0.61  0.30  0.17  0.12
Rene        1.43  1.32  1.15  1.24  1.13  0.88  0.87  0.89
Stacy       1.06  0.81  0.62  0.47  0.44  0.36  0.29  0.21
Tracy       1.51  1.14  1.02  0.73  0.56  0.55  0.59  0.59
true.
```

If no names are found, `switched` isn't very smart; it goes ahead and prints the header row:

```
?- switched(52,53).
            1952  1953
true.
```

If you want to make your `switched` smarter, that's fine—I won't test with any spans that produce no names. Also, I'll only test with spans where the first year is less than the last year.

Names are left-justified in a ten-wide field. Below is a `format` call that does that. Note that the dollar sign is included only to clearly show the end of the output.

```
?- format("~w~t~10|$", 'testing').
testing   $
true.
```

Outputting the ratios is a little more complicated. I'll spare you the long story but I use `sformat`, like this:

```
?- sformat(Out, '~t~2f~6|', 2.32754), write(Out).
```

```
  2.33
Out = "  2.33".
```

The call above instantiates `Out` to a six-character string (which is actually a list of ASCII character codes) and `write(Out)` outputs it.

See `help(format/2)` if you're curious about the details of using `~t` but the essence is that you can use `~N|` to specify that a field extend to column `N`, and then put a `~t` on the left, right, or both sides of a specifier like `~w` or `~f` to get right, left, or center justification, respectively.

To consult `a8/births.pl` when you consult `switched.pl`, put the following line in your `switched.pl`.

```
:-[a8/births].
```

That construction, `:-` followed by a query, causes the query to be executed when the file is consulted.

Note that `:-[a8/births.pl].` fails with an error. For an extra point on this assignment, add a note to `observations.txt` with a speculative but sound explanation of why `[a8/births]` works but `[a8/births.pl]` doesn't. No Googling, etc., please!

You might also use that `:- ...` mechanism to cause a couple of tests to be run when the file is loaded. Below I define `test/0` to do a couple of `switched` queries, putting a line of dashes between them. I then invoke it with `:-test.`

```
test :- switched(51,58), writeln('-----'), switched(51,52).

:-test.
```

That invocation of `test` must follow the definition of `switched` and consulting `a8/births.pl`.

You'll see that with `swipl -l switched` the output from `test` appears before "`Welcome to SWI-Prolog...`"

Be sure to comment out lines like `:-test.` before turning in your solution. (That output will cause `a8/tester` failures, too, as you'd expect.)

My Prolog solution is significantly smaller than my Ruby solution but it's easy to get sideways on this problem if you don't come up with a good set of helper predicates. I suggest that you give it a try on your own but if it starts to get ugly, http://www.cs.arizona.edu/classes/cs372/spring15/a8/switched-hints.pdf shows how I broke it down. The points assigned to this problem are based on the assumption that you will take a look at the hints. Without the hints, and based on your current level of Prolog knowledge, I might assign this problem 25-30 points.

Problem 6. (18 points) `iz.pl`

In this problem you are to write a predicate `iz/2` that evaluates expressions involving atoms and a set of operators and functions. Let's start with some examples:

```
?- S iz abc+xyz.        % + concatenates two atoms.
S = abcxyz.
```

```
?- S iz (ab + cd)*2.  % *N produces N replications of the atom.
S = abcdabcd.

?- S iz -cat*3.       % - is a unary operator that produces a reversed copy of the atom.
S = tactactac.

?- S iz -cat+dog.
S = tacdog.

?- S iz abcde / 2.    % / N produces the first N characters of the atom.
S = ab.

?- S iz abcde / -3.   % If N is negative, / produces last N characters
S = cde.

?- N is 3-5, S iz (-testing)/N.
N = -2,
S = et.
```

The functions `len` and `wrap` are also supported. `len(E)` evaluates to an **atom** (not a number!) that represents the length of `E`.

```
?- N iz len(abc*15).
N = '45'.

?- N iz len(len(abc*15)).
N = '2'.
```

`wrap` adds characters to both ends of its argument. If `wrap` is called with a two arguments, the second argument is concatenated on both ends of the string:

```
?- S iz wrap(abc, ==).
S = '==abc=='.

?- S iz wrap(wrap(abc, ==), '*' * 3).
S = '***==abc==***'.
```

If `wrap` is called with three arguments, the second and third arguments are concatenated to the left and right ends of the string, respectively:

```
?- S iz wrap(abc, '(', ')').
S = '(abc)'.

?- S iz wrap(abc, '>' * 2, '<' * 3).
S = '>>abc<<<'.
```

It is important to understand that `len(xy)`, `wrap(abc, ==)`, and `wrap(abc, '(', ')')` are simply structures. If `iz` encounters a two-term structure whose functor is `wrap` (like `wrap(abc, ==)`) its value is the concatenation of the second term, the first term, and the second term. `iz` evaluates `len` and `wrap` like `is` evaluates `random` and `sqrt`.

The atoms `comma`, `dollar`, `dot`, and `space` do not evaluate to themselves with `iz` but instead evaluate to the atoms `','`, `'$'`, `'.'`, and `' '`, respectively. (They are similar to `e` and `pi` in arithmetic expressions evaluated with `is/2`.) In the following examples note that `swipl` (not me!) is adding some

additional wrapping on the comma and the dollar sign that stand alone. That adornment disappears when those characters are used in combination with others.

```
?- S iz comma.
S = (',').

?- S iz dollar.
S = ($).

?- S iz dot.
S = '.'.

?- S iz space.
S = ' '.

?- S iz comma+dollar*2+space+dot*3.
S = ',$$ ...'.

?- S iz wrap(wrap(space+comma+space,dot),dollar).
S = '$. , .$'.

?- S iz dollarcommadotspace.
S = dollarcommadotspace.
```

The final example above demonstrates that these four special atoms don't have special meaning if they appear in a larger atom.

Here is a summary for `iz/2`:

> `-Atom iz +Expr` unifies `Atom` with the result of evaluating `Expr`, a structure representing a calculation involving atoms. The operators (functors) are as follows:
>
> | | |
> |---|---|
> | `E1+E2` | Concatenates the atoms produced by evaluating `E1` and `E2` with `iz`. |
> | `E*N` | Concatenates `E` (evaluated with `iz`) with itself `N` times. (Just like Ruby.) `N` is a term that can be evaluated with `is/2` (repeat, i**s**/2). Assume `N >= 0`. |
> | `E/N` | Produces the first (last) `N` characters of `E` if `N` is greater than (less than) 0. If `N` is zero, an empty atom is produced. (An empty atom is shown as two single quotes with nothing between them.) `N` is a term that can be evaluated with i**s**/2. The behavior is undefined if `abs(N)` is greater than the length of `E`. |
> | `-E` | Produces reversed `E`. |
> | `len(E)` | Produces an **atom**, not a number, that represents the length of `E`. |
> | `wrap(E1,E2)` | Produces `E2+E1+E2`. |
> | `wrap(E1,E2,E3)` | Produces `E2+E1+E3`. |

The behavior of `iz` is undefined for all cases not covered by the above. Things like `1+2`, `abc*xyz`, etc., simply won't be tested.

Here are some cases that demonstrate that the right-hand operand of `*` and `/` can be an arithmetic expression:

```
?- X = 2, Y= 3, S iz 'ab' * (X+Y*3).
X = 2,
Y = 3,
S = abababababababababab .

?- S = '0123456789', R iz S + -S, End iz R / -(2+3).
S = '0123456789',
R = '01234567899876543210',
End = '43210' .
```

Implementation notes

One of the goals of this problem is to reinforce the idea that for an expression like `-a+b*3` Prolog creates a tree structure that reflects the precedence and associativity of the operators. `is/2` evaluates a tree as an arithmetic expression. `iz/2` evaluates a tree as a "string expression". Note the contrast when the same tree is evaluated by `is` and `iz`:

```
?- X is pi + e*3.          % using is
X = 11.296438138966929.

?- X iz pi + e*3.          % using iz
X = pieee .
```

It's important to understand Prolog itself parses the expression and builds a corresponding structure that takes operator precedence into account. `display/1` shows the tree:

```
?- display(pi + e*3).
+(pi,*(e,3))
true.
```

Processing of syntactically invalid expressions like `abc + + xyz` never proceeds as far as a call to `iz`.

Below is some code to get you started. It fully implements the + operation.

```
% cat a8/iz0.pl
:-op(700, xfx, iz).    % Declares iz to be an infix operator. The leading :- causes the
                       % line to be evaluated as a goal, not consulted as a fact.

A iz A :- atom(A), !.
R iz E1+E2 :- R1 iz E1, R2 iz E2, atom_concat(R1, R2, R).
```

Here are examples that use the version of `iz` just above.

```
?- [a8/iz0].          % Note: no ".pl"  (or use ['a8/iz0.pl'])
% a8/iz0.pl compiled 0.00 sec, 3 clauses
true.

?- X iz abc+def.
X = abcdef.
```

```
?- X iz abc+def, Y iz X+'...'+X.
X = abcdef,
Y = 'abcdef...abcdef'.

?- X iz a+b+(c+(de+fg)+hij+k)+l.
X = abcdefghijkl.
```

Let's look at the code provided above. Let's first talk about the second clause:

```
R iz E1+E2 :- R1 iz E1, R2 iz E2, atom_concat(R1, R2, R).
```

The first thing you may notice is that the head doesn't match the *functor*(*term*,*term*,...) form we've always seen. That's because the `op(700, xfx, iz)` call above lets us use `iz` as an infix operator, and that applies in both the head and body of a rule. Here is a **completely equivalent version** that doesn't take advantage of the `op` specification:

```
iz(R,E1+E2) :- iz(R1,E1), iz(R2,E2), atom_concat(R1, R2, R).
```

With that equivalence now discussed, lets focus on the infix version:

```
R iz E1+E2 :- R1 iz E1, R2 iz E2, atom_concat(R1, R2, R).
```

Consider the goal `X iz ab+cd`. It unifies with the head of the above rule like this:

```
?- (X iz ab+cd) = (R iz E1+E2).
X = R,
E1 = ab,
E2 = cd.
```

The first goal in the body of the rule is `R1 iz ab`, observing that `E1` is instantiated to `ab`. That goal unifies with the head of this rule:

```
A iz A :- atom(A), !.
```

This rule represents the base case for the recursive evaluation performed by `iz`. It says, "If `A` is an atom then the result of evaluating that atom is `A`." Another way to read it is, "An atom evaluates to itself." The result is that `R1 iz ab` instantiates `R1` to `ab`. Note that atoms always lie at the leaves of the expression's tree.

It's important to recognize that because the `iz(R, E1+E2)` rule is recursive, it'll handle every tree composed of + operations.

Here are the heads for all the `iz` rules that I've got in my solution:

```
R iz E1+E2 :-

R iz E1*NumExpr :-

R iz -E :-

R iz E1 / NumExpr :-

R iz len(E) :-

R iz wrap(E,Wrap) :-
```

```
R iz wrap(E,First,Last) :-
```

Via recursion these heads handle all possible combinations of operations, like this one:

```
?- X iz wrap((-(ab+cde*4)/6+xyz), 'Start>','<'+(end*3+zz*2)).
X = 'Start>edcedcxyz<endendendzzzz'.
```

If you find yourself wanting to write rules like `R iz (E1+E2) * NumExpr) :- ...` then STOP! You're probably not recognizing that rules based on the above cover everything.

On the slides I fail to mention that Prolog requires some sort of separation between operators. Consider this:

```
?- X iz abc+-abc.     % No space between + and -
ERROR: Syntax error: Operator expected
ERROR: X iz abc
ERROR: ** here **
ERROR: +-abc .
```

To make it work, add a space or parenthesize:

```
?- X iz abc+ -abc.
X = abccba.

?- X iz abc+(-abc).
X = abccba.
```

This issue only arises with unary operators, of course.

Problem 7. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your hours on a line that starts with `"Hours:"`. Some students are including per-problems times, too. That's useful and interesting data—keep it coming!—but `observations.txt` should have only one line that starts with `Hours:`. If you care to report per-problem times, impress me with a good way to show that data.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use the D2L Dropbox named `a8` to **submit a single zip file named `a8.zip` that contains all your work**. If you submit more than one `a8.zip`, your final submission will be graded. Here's the full list of deliverables:

`splits.pl`
`repl.pl`
`pick.pl`
`polyperim.pl`
`switched.pl`
`iz.pl`
`observations.txt` (for extra credit)

Note that all characters in the file names are lowercase.

Miscellaneous

Here's what `wc` shows for my current solutions, which are typically written with one goal per line for the longer predicates.

```
$ wc splits.pl repl.pl pick.pl polyperim.pl switched.pl iz.pl
   1    9   65 splits.pl
   2   13   86 repl.pl
   4    8  169 pick.pl
  16   43  482 polyperim.pl
  41   79 1081 switched.pl
  47  156  983 iz.pl
 111  308 2866 total
```

You can use any elements of Prolog that you desire other than if-then-else (`->`) and disjunction (`;`), but the assignment is written with the intention that it can be completed easily using only the material presented on Prolog slides 1-160.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem on this assignment corresponds to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.) In Prolog, a % is comment to end of line. `/* ... */` can be used for block comments, just like in Java.

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 6 to 8 hours to complete this assignment.

Keep in mind the point value of each problem; don't invest an inordinate amount of time in a problem or become incredibly frustrated before you ask for a hint or help. Remember that the purpose of the assignments is to build understanding of the course material by applying it to solve problems. If you reach the six-hour mark, regardless of whether you have specific questions, it's probably time to touch base with me. Give me a chance to speed you up! **My goal is that everybody gets 100% on this assignment AND**

gets it done in an amount of time that is reasonable for them.

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, and more. (See the syllabus for the details.)